Trace-Based Aspects

Rémi Douence, Pascal Fradet, Mario Südholt

Trace-based Aspects

Rémi Douence[1,*], Pascal Fradet[2], Mario Südholt[1,*]

[1]École des Mines de Nantes/INRIA, Nantes, France
www.emn.fr/{douence,sudholt}
[2]IRISA/INRIA, Rennes, France
www.irisa.fr/lande/fradet

Abstract

In this article, we present trace-based aspects which take into account the history of program executions. They are defined in terms of execution traces and may express relations between different events. Weaving is modeled by an execution monitor which modifies the base program execution as defined by the aspects. We motivate trace-based aspects and explore options within the trade-off between expressiveness and property enforcement/analysis.

More concretely, we first present a very expressive model of trace-based aspects enabling proofs of aspect properties by equational reasoning. Using a restriction of the aspect language to regular expressions, we show that it becomes possible to address the difficult problem of interactions between conflicting aspects. Finally, by restricting the actions performed by aspects, we illustrate how to keep the semantic impact of aspects under control and to implement weaving statically.

1 Introduction

Aspect-Oriented Programming (AOP) is concerned with providing programmatic means to modularize crosscutting functionalities of complex applications. By encapsulating such functionalities into aspects, AOP intends to facilitate development, understanding and maintenance of programs. An important characteristics of aspects is that they are built from *crosscuts* (pointcuts in ASPECTJ), which define where an aspect modifies an application, and *inserts* (advice in ASPECTJ), which define the modifications to be applied. Typically, crosscuts denote sets of

*Partially funded by the EU project "EasyComp" (www.easycomp.org), no. IST-1999-014191.

program points or execution points of the base application and inserts are expressed in a traditional programming language. For instance, an aspect for access control could be defined in terms of crosscuts denoting sets of access methods and inserts performing the necessary tests. However, because aspect languages are rather restricted, it is often necessary to use inserts to pass information from one crosscut to another one. Consider, for instance, an aspect performing some access control after some login event. When the aspect language cannot take into account the history of computation, an insert must be used to set a flag when a login takes place. By testing the flag, further crosscuts know whether access control should be performed or not.

The fact that inserts are unrestricted and that their role overlaps with the role of crosscuts, makes reasoning on aspects and woven programs difficult. In this article, we present an approach which — by means of expressive aspect languages and restrictions on inserts — enable reasoning about different aspect properties.

Trace-Based Aspects are defined on traces of events occurring during program execution. Trace-based aspects are more expressive than those based on atomic points because relations between execution events – possibly involving information from the corresponding execution states — can be expressed. For example, an aspect for access control could express that a user has to log in first in order to pass an access check later. Such aspects are called *stateful*: their implementation must use some kind of state to represent their evolution according to the event encountered. Conceptually, weaving is modeled by an execution monitor whose state evolves according the history of program execution and which, in case of a match, triggers the execution of the corresponding action. By strictly separating crosscuts and inserts by means of two different, well-defined languages, we address the formalization of aspects and weaving. Restrictions on these languages allow us to design static analysis of aspect properties as well as an optimized implementation of aspect weaving.

We first (Section 2) introduce informally the main features of trace-based AOP: observable execution traces, stateful aspects (composed of crosscuts and inserts), and weaving (based on execution monitoring). In Sections 3–5, we explore three different options within the trade-off between expressiveness and property enforcement/analysis. The first option provides a very expressive crosscut language and does not impose restrictions on the inserts. Due to its expressive power, however, only manual proofs of aspect properties, e.g. equivalence of aspects, are supported in this case. The second option is characterized by more restricted, but still stateful, aspects corresponding to regular expressions over execution traces. Because of this restriction it is possible to statically detect whether several aspects interact (*e.g.*, testing whether an encryption aspect interacts with a system logging aspect). We also suggest operators for the resolution of such interactions. The last option

is characterized by a very restricted insert language where aspects can be seen as formal (safety) properties. We present how these aspects/properties can be statically and efficiently woven. An application of this technique is the securization of mobile code upon receipt. Finally, we discuss related work and conclude in Section 6.

This article is a unified presentation of three distinct studies [DMS01, DFS02, CF00] sharing a trace-based approach to AOP. In order to make the presentation more intuitive, we have deliberately omitted many extensions and technical details. The interested reader may find them in the original conference papers.

2 Characteristics of Trace-Based Aspects

Trace-based aspects have two main characteristics. First, aspects are defined over sequences of observable execution states. Second, weaving is performed on executions rather than program code. The weaver can be seen as a monitor interleaving the execution of the base program and execution of inserts[1].

2.1 Observable execution trace

The base program execution is modeled by a sequence of observable execution states (a.k.a. join points). This trace can be formally defined on the basis of the small-step semantics [NN92] of the programming language. Each join point is an abstraction of the execution state. Join points may denote syntactic information (e.g., instructions) but also semantic one (e.g., dynamic values). For example, when the user Bob logs, the function `login()` is called in the base program with `"Bob"` as a parameter. This join point of the execution can be represented by the term `login("Bob")`.

2.2 Aspect language

The basic form of an aspect is a rule of the form $C \triangleright I$ where C is a crosscut and I is an insert. The insert I is executed whenever the crosscut C matches the current join point. Basic aspects can be combined using operators (sequence, repetition, choice, etc.) to form stateful aspects.

Crosscuts. A crosscut defines execution points where an aspect should perform an action. In general, a crosscut C is a function that takes a join point as a parameter. This function returns either `fail` when the join point does not satisfy the cross-

[1] Note that this model does not prevent weaving to be a compile-time process (see Section 5)

cut definition, or a substitution that captures values of the join point. For example, we can define a crosscut *isLogin* that matches session logins and captures the corresponding user name. It would return `fail` when it is applied to the join point `logout()` and the substitution $uid = $ `"Bob"` when it is applied to `login("Bob")`.

Inserts. An insert is an executable program fragment with free variables. For instance, the insert `addLog(`*uid* `+ "logged in")` prints the name of a logged user when it is executed. In this insert, the name of the user is represented by the variable *uid* to be bound by a crosscut. In the remainder of the paper, the special insert `skip` represents an instruction doing nothing.

Stateful aspects. The intuition behind a basic aspect $C \triangleright I$ is that when C matches the current join point and yields a substitution ϕ, the program $\phi\, I$ is executed. For example, we can define a basic security aspect which logs sessions as follows:

$$\textit{isLogin} \triangleright \texttt{addLog(}\textit{uid}\texttt{ + "logged in")}$$

In order to build stateful aspects, basic aspects can be combined using control operators. Using a C-like syntax, we can define an aspect which logs all sessions as follows:

$$\texttt{while(true)\{}\ \ \textit{isLogin} \triangleright \texttt{addLog(}\textit{uid}\texttt{ + "logged in");}\ \ \texttt{\}}$$

This definition applies the basic security aspect again and again. Control operators allow us to define sophisticated aspects on execution traces. For instance, the following aspect tracks sequences of sessions (`login` followed by `logout`).

$$\begin{array}{l}\texttt{while(true)\{}\ \ \textit{isLogin} \triangleright \texttt{addLog(}\textit{uid}\texttt{ + "logged in");}\\ \qquad\qquad\quad\ \ \textit{isLogout} \triangleright \texttt{addLog(}\textit{uid}\texttt{ + "logged out");}\ \ \texttt{\}}\end{array}$$

2.3 Weaving

In general, several aspects addressing different issues (e.g., debugging and profiling) can be composed (using a parallel operator $\|$) and woven together. The weaver takes a parallel composition of n aspects $A_1 \| \ldots \| A_n$ and tries to apply each of them (in no specific order) at each join point of the execution trace.

Conceptually, the weaver is an execution monitor that selects the current basic aspects of $A_1, \ldots, A_n$ and tries to apply them at each join point. When a crosscut matches the current join point, the corresponding insert is executed. After all basic aspects have been considered, the base program execution is resumed and proceeds until the next join point.

When a basic aspect of a stateful aspect A_i has been applied and its insert executed, the state of A_i evolves. The control structure of A_i (e.g. repetition or sequence) specifies which basic aspect must be considered next. For instance, the previous security aspect remains in its initial state until a login occurs. After the aspect has matched a `login` event, it waits to match a `logout` event before returning to its initial state.

In the remainder of this article, this framework is instantiated in form of different definitions of crosscuts, inserts and stateful aspects. We thus obtain different aspect languages which can be used to reason about aspect-oriented programs (manually or using static analysis techniques).

3 Expressive aspects and equational reasoning

We now present a first instantiation of the general framework for AOP introduced in the previous section. This instantiation, which is inspired by the work presented in [DMS01], is intended to illustrate two main points:

- The usefulness of expressive aspect definitions.
- The application of general proof techniques for the analysis and transformation of AO programs.

Crosscuts In this section we instantiate the general framework by allowing crosscuts C to be arbitrary predicates. For instance, a predicate *isWeakPassword* could discriminate events occurring when a password should be changed to a word which belongs to a dictionary. Note that we do not define the crosscuts we use in this section; they are supposed to be defined using some general-purpose language.

Stateful Aspects. Since one of our main interests lies in the definition of *stateful* crosscuts, we base aspect definitions on the following grammar:

$$
\begin{array}{rcll}
A & ::= & C \rhd I & \textit{; basic aspect} \\
 & | & A_1 \,;\, A_2 & \textit{; sequence} \\
 & | & A_1 \,\Box\, A_2 & \textit{; choice} \\
 & | & \mu a.A & \textit{; recursive definition} \\
 & | & a & \textit{; recursive call}
\end{array}
\qquad (1)
$$

This grammar allows us to compose complex aspects by recursion, sequentialization and deterministic choice ($A_1 \Box A_2$ chooses A_1 if both aspects A_1 and A_2 are applicable at the current join point). Using composed aspects, we can define, for

example, an aspect trying to apply $C \triangleright I$ only on the current join point and doing nothing afterward as

$$(C \triangleright I\,;\,(\mu a.\mathit{isAny} \triangleright \texttt{skip}\,;\,a)) \,\square\, (\mu a.\mathit{isAny} \triangleright \texttt{skip}\,;\,a)$$

If C matches the current join point, the weaver chooses the first branch, executes the insert I and the aspect becomes $\mu a.\mathit{isAny} \triangleright \texttt{skip}\,;\,a$ that keeps doing nothing. Otherwise, the weaver chooses the second branch which keeps doing nothing right from the start.

In order to illustrate how such expressive aspects may be used consider the following definition:

$$\begin{aligned}\mathit{logNestedLogin} = \mu a_1.\mathit{isLogin} \triangleright \texttt{skip}\,;\\ (\mu a_2.\mathit{isLogin} \triangleright \texttt{addLog(}\mathit{uid}\texttt{)}\,;\,a_2\,;\,\mathit{isLogout} \triangleright \texttt{skip}\\ \square\ \mathit{isLogout} \triangleright \texttt{skip})\,;\,a_1\end{aligned}$$

The aspect *logNestedLogin* considers sessions starting with a call to the `login` function with the user identifier as a parameter (as defined by the crosscut *isLogin*) and ending with a call to the function `logout` (as defined by the crosscut *isLogout*). This aspect logs nested (i.e. non top-level) calls to the login function because such a call may login into a non-local network and be therefore dangerous. Note that there is no need of a stack or an integer counter in inserts to take into account nested sessions. The recursive definition of the aspect is responsible for pairing logins and logouts, thus detecting non top-level calls to `login`.

Now, let us consider the following aspect:

$$\mathit{initAtFirstLogin} = \mathit{isLogin} \triangleright \texttt{initNetworkInfo}()\,;\,\mu a.\mathit{isLogin} \triangleright \texttt{skip}\,;\,a$$

This aspect *initAtFirstLogin* detects the first call to login in order to initialize network information. Then the following calls to login are ignored. Note that we chose simple examples for demonstration purposes.

It is easy to prove that the two aspects *logNestedLogin* and *initAtFirstLogin* are equivalent to a single sequential aspect. Basically, this can be proven by unfolding of recursive definitions and induction principles [DMS01]. The proof starts with a parallel composition *logNestedLogin* $\|$ *initAtFirstLogin* and eliminates the parallel operator by producing all the possible pairs of crosscuts from the two aspect definitions and by folding. The resulting sequential aspect can be simplified if a pair of crosscut has no solution. In our example we get the following sequential aspect:

$$
\begin{aligned}
\mathit{initAndLog} = {} & \mathit{isLogin} \triangleright \texttt{initNetworkInfo();} \\
& (\mu a_2.\mathit{isLogin} \triangleright \texttt{addLog(}\mathit{uid}\texttt{)}; a_2; \mathit{isLogout} \triangleright \texttt{skip} \\
& \quad \Box\ \mathit{isLogout} \triangleright \texttt{skip}); \\
& \mu a_3.\mathit{isLogin} \triangleright \texttt{skip}; \\
& (\mu a_4.\mathit{isLogin} \triangleright \texttt{addLog(}\mathit{uid}\texttt{)}; a_4; \mathit{isLogout} \triangleright \texttt{skip} \\
& \quad \Box\ \mathit{isLogout} \triangleright \texttt{skip}); a_3
\end{aligned}
$$

By restricting the expressiveness of our aspect language (while still adhering to *stateful* aspects), it is possible to automatically prove (certain) aspect properties. This is the subject of the following section.

4 Detection and resolution of aspect interactions

In this section we consider a second instantiation of the general framework that supports a more restrictive yet expressive crosscut language in which static checking of interactions is feasible.

Crosscuts. A crosscut is defined by conjunctions, disjunctions and negations of terms:

$$C ::= T \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid \neg C \tag{2}$$

where T denotes terms with variables. The formulas used to express these crosscuts belong to the so-called quantifier-free equational formulas [Com91]. Whether such a formula has a solution is decidable. This is one of the key properties making the analysis in this section feasible.

We can define, for example, a crosscut matching logins performed by the user `root` on any machine, or by any non-root user on any machine but the `server` as follows:

$$\texttt{login(root,}m\texttt{)} \vee (\texttt{login(}u\texttt{,}m\texttt{)} \wedge \neg \texttt{login(}u\texttt{, server)})$$

In this context, checking whether the current join point (which is represented, remember by a term) matches the crosscut definition is computed by a generalized version of the unification algorithm.

Note that, for the sake of decidability (i.e. static analyses) the crosscuts defined by Equation 2 are less expressive than those considered in the previous section. They can only denote join points as term patterns (as opposed to arbitrary term predicates).

Stateful aspects. The main idea of the aspect language presented in this section is to restrict stateful aspects to regular expressions using the following grammar:

$$\begin{array}{rcll} A & ::= & \mu a.A & \textit{; recursive definition} \\ & | & C \rhd I\,;A & \textit{; sequence} \\ & | & C \rhd I\,;a & \textit{; end of sequence} \\ & | & A_1 \,\square\, A_2 & \textit{; choice} \end{array}$$

Using this aspect language a security aspect that logs file accesses (calls to `read`) during a non-nested session (from a call to `login` until a call to `logout`) can be expressed as

$$\begin{array}{l} \textit{logAccess} = \\ \quad \mu a_1.\texttt{login}(u,m) \rhd \texttt{skip}\,; \\ \qquad (\mu a_2.(\texttt{logout}() \rhd \texttt{skip}\,;a_1) \,\square\, (\texttt{read}(x) \rhd \texttt{addLog}(x)\,;a_2)) \end{array} \tag{3}$$

where x matches the name of the accessed file.

4.1 Aspect interactions

Remember that a parallel composition of n aspects $A_1 \| \ldots \| A_n$ does not define any specific order of application of aspects; so the result of the weaving process may be non-deterministic. This situation arises when aspects interact, that is to say when at least two inserts must be executed at the same join point. For instance, let us consider the following aspect:

$$\textit{cryptRead} = \mu a.\texttt{read}(x) \rhd \texttt{crypt}(x)\,;a$$

This aspect states that the reads should be encrypted. It obviously interacts with the aspects defined in Equation 3 which describes logging for all users. When a user logs in and accesses a file, this access must be logged *and* the file name must be encrypted.

The algorithm to check aspects interaction is similar to the algorithm for finite-state product automata. It terminates due to the finite-state nature of our aspects. Starting with a composition $A\|A'$, the algorithm eliminates the parallel operator by producing all the possible pairs (conjunction) of crosscuts from A and A'. A pair of crosscuts is a solvable formula and we can check if it has a solution using the algorithm of [Com91]. A pair of crosscuts with no solution cannot match any join point and can be removed from the aspect (for details see [DFS02]). In the case of the previous example *logAccess* $\|$ *cryptRead*, we get:

logAccess $\|$ *cryptRead* =

$\mu a_1.\texttt{login}(u,m) \rhd \texttt{skip}\,;$
$(\mu a_2.(\texttt{logout}() \rhd \texttt{skip}\,; a_1)$
$\Box\ (\texttt{read}(x) \rhd (\texttt{addLog}(x) \bowtie \texttt{crypt}(x))\,; a_2)$
$\Box\ \texttt{read}(x) \rhd \texttt{crypt}(x)\,; a_1$

Conflicts are represented using the non-deterministic function $(I_1 \bowtie I_2)$ which returns either $I_1;I_2$ or $I_2;I_1$. Here, we have $(\texttt{addLog}(x) \bowtie \texttt{crypt}(x))$, so the two aspects are not independent. Note that spurious conflicts have already been eliminated with the help of the rule $(I \bowtie \texttt{skip}) = (\texttt{skip} \bowtie I) = I$.

This analysis does not depend on the base program to be woven. When there is no $(\bowtie)$ in the resulting sequential aspect, the two aspects are independent for all programs. This property does not have to be checked again after each program modification. However, this property is a sufficient but not a necessary condition. A more precise analysis is possible by taking into account the possible sequences of join points generated by the base program to be woven (see [DFS02]).

4.2 Support for conflict resolution

When no conflict have been detected, the parallel composition of aspects can be woven without modifications. Otherwise, the programmer must get rid of the non-determinism by making the composition more precise. We present in the following some linguistic support aimed at resolving interactions.

The occurrences of rules of the form $C \rhd (I_1 \bowtie I_2)$ indicate all potential interactions. They can be resolved one by one. For each $C \rhd (I_1 \bowtie I_2)$, the programmer may replace each rule $C \rhd (I_1 \bowtie I_2)$ by $C \rhd I_3$ where I_3 is a new insert which combines I_1 and I_2 in some way. For instance, in the previous example, $(\texttt{addLog}(x) \bowtie \texttt{crypt}(x))$ can be replaced by $\texttt{crypt}(x);\texttt{addLog}(x)$ in order to generate encrypted logs.

This option is flexible but can be tedious. Instead of writing a new insert for each conflict, the programmer may indicate how to compose inserts at the aspect level. We propose a parallel operator $\|_{seq}$ to indicate that whenever a conflict occurs, $(I_1 \bowtie I_2)$ must be replaced by $I_1;I_2$ (where ";" denotes the sequencing operator of the programming language). Other parallel operators are useful, such as $\|_{fst}$ which replaces $(I_1 \bowtie I_2)$ by I_1 only.

Let us reconsider the two aspects *logAccess* and *cryptRead*.

- *logAccess* $\|_{seq}$ *cryptRead* generates plaintext logs for super users,
- *cryptRead* $\|_{seq}$ *logAccess* generates logs for users by logging (possibly encrypted) accesses,

5 Static weaving of safety properties

The previous restrictions allowed us to detect interactions during weaving. However, they are not sufficient to detect semantic interactions. The code inserted by an aspect may still influence the application of another independent aspect. Our notion of independence only ensures that aspects can be woven in any order. In order to prevent semantic interactions and, more generally, to control the semantic impact of weaving, one has to restrict the language of inserts. Here, we consider the same aspect language as the previous section, except for the language of inserts which becomes

$$I ::= \texttt{skip} \mid \texttt{abort}$$

Even if this restriction is quite drastic (aspects can only abort the execution), interesting aspects can still be expressed. The expressive crosscut language allows us to specify safety properties (properties stating that no "bad thing" happens during the execution). Aspects can be used to rule out unwanted execution traces and to express security policies [CF00].

This restriction has several benefits:

- Aspects are semantic properties and the impact of weaving is clear.
- Inserts always commute; there are no interactions between aspects which can be composed in parallel.

The woven program satisfies the property/aspect: for executions in accordance with the property, it has the same behavior as the base program; otherwise, it produces an exception and terminates just before violating the property.

The main drawback of execution monitors is their runtime cost. They are not specialized to the program and each program instruction may involve a runtime check. In the remainder of this section we present how to weave such trace-based aspects statically and efficiently.

5.1 Example

Consider the following aspect

$$\begin{array}{l}
\mu a.\texttt{accountant()} \triangleright \texttt{skip;} \big(\texttt{manager()} \triangleright \texttt{skip;} \texttt{critical()} \triangleright \texttt{skip;} a \\
\qquad\qquad\qquad\qquad\quad \Box\ \texttt{critical()} \triangleright \texttt{abort;} a \big) \\
\Box\ \texttt{manager()} \triangleright \texttt{skip;} \quad \big(\texttt{accountant()} \triangleright \texttt{skip;} \texttt{critical()} \triangleright \texttt{skip;} a \\
\qquad\qquad\qquad\qquad\quad \Box\ \texttt{critical()} \triangleright \texttt{abort;} a \big) \\
\Box\ \texttt{critical()} \triangleright \texttt{abort;} a
\end{array}$$

The property defined by the aspect states that a critical action cannot take place before the clearance of the manager and the accountant (at least a call to `manager` and to `accountant` must occur before each call to `critical`).

Fig. 1 illustrates weaving of this property on a very simple imperative base program. Since the property is specify by a finite state aspect, it can be encoded as an automaton with alphabet {`m, a, c`} corresponding to the calls to `manager`, `accountant` and `critical` respectively. Notice that the base program may violate this property whenever the condition of the first `if` statement is false. The woven program, where two assignments and a conditional have been inserted, satisfies the property (i.e. aborts whenever the property is about to be violated).

An important challenge is to make this dynamic enforcement as inexpensive as possible. In particular, if we are able to detect statically that the base program satisfies the property, then no transformation should be performed.

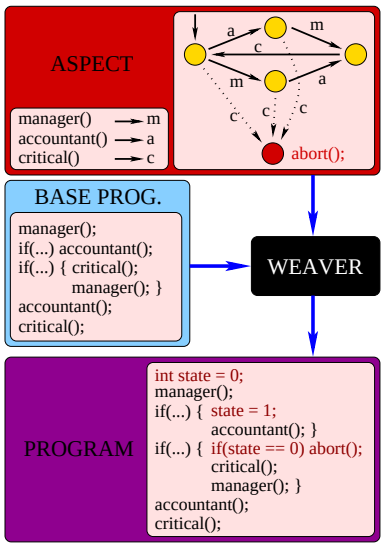

Figure 1: *A simple example*

5.2 Weaving phases

Our aspects define a regular set of allowed finite executions. An aspect is encoded as a finite state automaton over events. The language recognized by the automaton is the set of all authorized sequences of events.

The weaver is a completely automatic tool which takes the automaton, the base program and produces an instrumented program [CF00]. We now outline its different phases (depicted in Fig. 2).

Base Program annotation.

The first phase is to locate and annotate the instructions of the base program corresponding to events (crosscuts). Depending on the property we want to enforce, the events can be calls to specific methods, assignments to specific variables, opening of files, etc. A key constraint is that an instruction of the base program must be associated with at most one event. This is easy to ensure when events are specified solely based on the syntax. In order to take semantic crosscuts into account, the base program must be transformed beforehand. Consider the event "*x is assigned the value* 0", it cannot be statically decided whether an assignment `x:=e` will generate this event or not. A solution is to transform each assignment `x:=e`

into the statement `if e=0 then x:=e else x:=e` where each instruction is now associated with a single event. Such pre-transformations rely on static program analyses to avoid insertion of useless tests.

Base Program abstraction.

The base program is abstracted into a graph whose nodes denote program points and edges represent instructions (events). The abstraction makes the next two phases independent of a specific programming language. Since the aspect is a trace property, the abstraction is the control-flow graph of the base program. In order to produce a precise abstraction, this phase relies on a control-flow analysis.

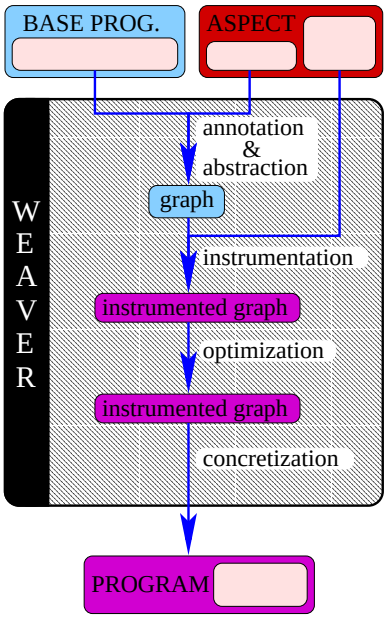

Figure 2: *Weaving phases*

Instrumentation. The next phase is to transform the graph in order to rule out the forbidden sequences of events. We integrate the automaton by instrumenting the graph with additional structures (states and transition functions) that mimic the evolution of the automaton. Intuitively, this instrumentation corresponds to the insertion of an assignment (to implement the state transition of the underlying automaton) and a test (to check whether the property is about to be violated) before each event. This naive weaving is optimized by the next phase.

Optimizations. The instrumented graph is refined in three steps. First, the automaton specifies a general property independent of any particular program. The first step is to *specialize* the automaton with respect to the base program. Second, the second step yields a normalized instrumented graph using a transformation similar to the classical automaton *minimization*. Finally, the last optimization removes useless state transitions using static analyses.

The graph after optimization represents a program where at most one test and/or assignment (state transition) have been inserted at each `if` and `while` statement.

Concretization. The optimized graph must be translated back into a program. The graph has remained close to the base program since its nodes and edges still represent the same program points and instructions. We just need a way to store, fetch, and test a value (the automaton state) without affecting the normal execution. This can be done by local transformations (e.g. inserting assignments and conditionals

on a fresh global variable).

5.3 Just-in-time weaving

The most interesting application of this technique is the securization of mobile code upon receipt. The local security policy is declared as a property (aspect) to be enforced on incoming applets. The just-in time weaver securizes (i.e. abstracts, instruments, optimizes, and concretizes) an applet before loading it. Since some steps are potentially costly, our implementation uses simple heuristics that make the time complexity of weaving linear in the size of the program.

There are several benefits to this separation of security concerns. First, it is easier to express the policy declaratively as a property. Second, the approach is flexible and can accommodate customized properties. This feature is especially important in a security context where it is impossible to foresee all possible attacks and where policies may have to be modified quickly to respond to new threats.

6 Conclusion

In this article, we have presented a model (and three instantiations) for AOP based on execution traces. We have focused on the following points:

- Expressive and stateful aspect definitions.
- A model conceptually based on weaving of executions.
- Reasoning about and analysis of aspect properties, in particular aspect interaction.
- Enforcement of properties by program transformation, i.e. static weaving.

We now briefly consider these contributions in turn and compare our approach with related work.

We have advocated and presented expressive (i.e. stateful) aspect languages. The crosscut language of ASPECTJ [K+01] consists mostly in *single instruction patterns* matching events such as a method calls or field accesses. ASPECTJ's patterns are very similar to our basic aspects $C \triangleright I$. Expressing *stateful* aspects in ASPECTJ requires book-keeping code in advice to pass information between crosscuts (e.g., increment a counter in an advice to check for the counter value later). The ASPECTJ construction `cflow` is the only exception allowing the definition of a form of stateful aspect. For example, `cflow(call(critical)) && call(read)` matches join points where `read` is called whenever there is a pending call to `critical` in the execution stack.

Our model is conceptually based on a monitor observing and weaving execution traces. Other techniques can be related to execution monitors. Computational reflection is a general technique used to modify the execution mechanisms of a programming language. Restricted approaches to reflection have been proposed in order to support AOP. For instance, the composition filter model [BA01] proposes method wrappers in order to filter method calls and returns. DeVolder *et al.* [Vol99] propose a meta-programming framework based on Prolog. Unfortunately, these approaches do not allow stateful aspects.

By appropriate restrictions of the aspect language we have proposed some solutions to the difficult problem of aspect interactions. Recent releases of ASPECTJ provide limited support for aspect interaction analysis using IDE integration: the base program is annotated with crosscutting aspects. This graphical information can be used to detect conflicting aspects. However, the simple (i.e. stateless) crosscut model of ASPECTJ would entail an analysis detecting numerous spurious conflicts because the book-keeping code cannot be taken into account. In case of real conflicts, ASPECTJ programmers must resolve conflicts by reordering aspects using the keyword `dominate`. When two aspects are unrelated *w.r.t.* the domination or hierarchy relations, the ordering of inserts is undefined.

In order to define static interaction analysis we had to formally define aspects and weaving (see [DFS02] for a formal treatment of Section 4). There are several approaches to the formalization of AOP. Wand *et al.* [WKD02] propose a denotational semantics for a subset of ASPECTJ. Lämmel [Läm02] formalizes method-call interception with big-step semantics. Andrews' model [And01] relies on algebraic processes. He focuses on equivalence of processes and correctness (termination) of the weaving algorithm.

Finally, we have shown in Section 5 that by restricting the insert language, aspects can be seen as formal properties which can be enforced by program transformation. Dynamic monitors (such as VeriSoft [God97] and AMOS [CFNF97]) or "security kernels" (such as Schneider's security automata [Sch00]) have been used to enforce security properties. By contrast, our programming language approach permits many optimizations and avoids to extend the runtime system or the language semantics.

The different aspect languages presented suggest several extensions. For example, allowing crosscuts of the same aspect to share variables would make the aspect language more expressive. The possibility of associating an instance of an aspect with a run-time entity (e.g. each instance of a class in a Java program) would facilitate the application of our model to object-oriented languages. It would been interesting to characterize a larger class of inserts (beyond `abort`) allowing to keep the semantic impact of weaving under strict control. More generally, we believe that an important avenue for further AOP research is to provide more safeguards in

terms of static analyses and specially-tailored aspect languages.

References

[And01] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Reflection*, pages 187–209, 2001.

[BA01] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[CF00] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 54–66, N.Y., January 19–21 2000. ACM Press.

[CFNF97] D. Cohen, M. S. Feather, K. Narayanaswamy, and S. S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 602–603. ACM Press, 1997.

[Com91] H. Comon. Disunification: A survey. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 322–359, 1991.

[DFS02] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, October 2002.

[DMS01] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'01)*, volume 2192 of *LNCS*. Springer Verlag, September 2001.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, 15–17 January 1997.

[K+01] G. Kiczales et al. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[Läm02] R. Lämmel. A semantics for method-call interception. In *1st Int. Conf. on Aspect-Oriented Software Development (AOSD'02)*, April 2002.

[NN92] F. Nielson and H. R. Nielson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, New York, NY, 1992.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[Vol99] K. De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.

[WKD02] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *FOOL 9*, pages 67–88, January 2002.